Design and Analysis of Algorithms (21CS42)

MODULE 5-

Backtracking
Branch and bound

General method

Backtracking is one of the most general techniques. many problems which deal with searching for a set of solutions or which ask for a optimal solution satisfying some constraints can be solved using the backtracking formulation. The name backtracking was first coined by D.H. Lehmer in 1950s.

N-Queens problem

The problem is to place n queens on an $n \times n$ chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal. For $n = 1$, the problem has a trivial solution, and it is easy to see that there is no solution for $n = 2$ and $n = 3$. So let us consider the four-queens problem and solve it by the backtracking technique. Since each of the four queens has to be placed in its own row, all we need to do is to assign a column for each queen on the board presented in Figure 1.

We start with the empty board and then place queen 1 in the first possible position of its row, which is in column 1 of row 1. Then we place queen 2, after trying unsuccessfully columns 1 and 2, in the first acceptable position for it, which is square (2, 3), the square in row 2 and column 3. This proves to be a dead end because there is no acceptable position for queen 3. So, the algorithm backtracks and puts queen 2 in the next possible position at (2, 4). Then queen 3 is placed at (3, 2), which proves to be another dead end. The algorithm then backtracks all the way to queen 1 and moves it to (1, 2). Queen 2 then goes to (2, 4), queen 3 to (3, 1), and queen 4 to (4, 3), which is a solution to the problem. The state-space tree of this search is shown in Figure 5.2. If other solutions need to be found the algorithm can simply resume its operations at the leaf at which it stopped.

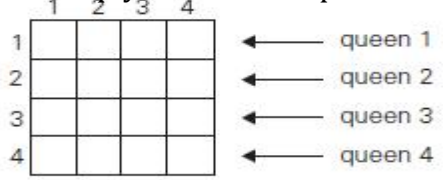

FIGURE 1 Board for the four-queens problem.

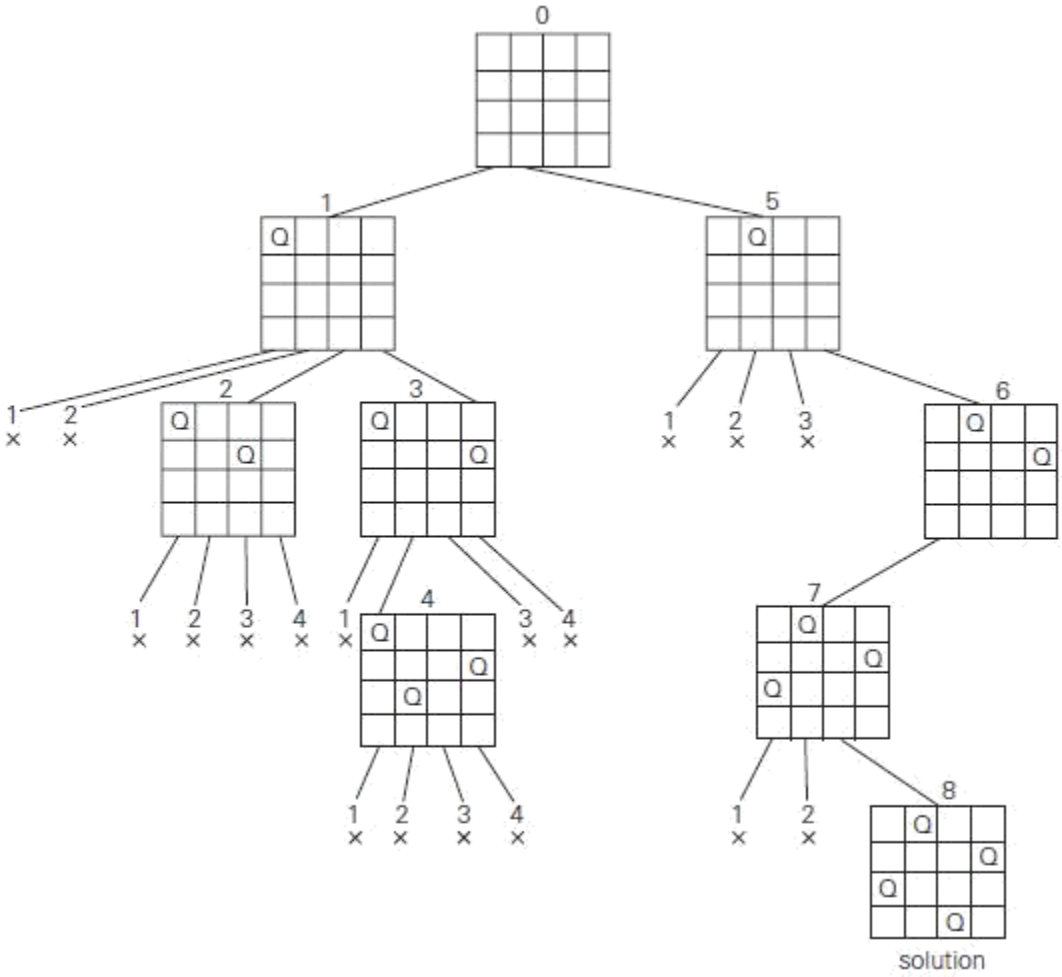

FIGURE 2 State-space tree of solving the four-queens problem by backtracking.

× denotes an unsuccessful attempt to place a queen in the indicated column. The numbers above the nodes indicate the order in which the nodes are generated. Finally, it should be pointed out that a single solution to the *n*-queens problem for any $n \geq 4$ can be found in linear time. In fact, over the last 150 years mathematicians have discovered several alternative formulas for non attacking positions of *n* queens .

Sum of subsets problem

the *subset-sum problem*: find a subset of a given set $A = \{a1, \ldots, an\}$ of *n* positive integers whose sum is equal to a given positive integer *d.* For example, for $A = \{1, 2, 5, 6, 8\}$ and $d = 9$, there are two solutions: {1, 2, 6} and {1, 8}. Of course, some instances of this problem may have no solutions.

It is convenient to sort the set's elements in increasing order. So, we will assume that $a1 < a2 < \ldots < an$.

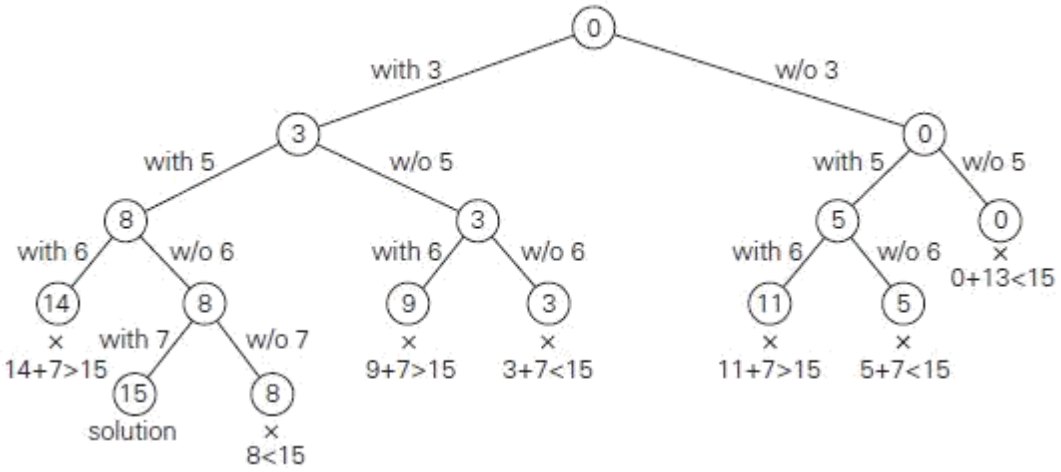

Figure 3 Complete state-space tree of the backtracking algorithm applied to the

instance $A = \{3, 5, 6, 7\}$ and $d = 15$ of the subset-sum problem. The number inside a node is the sum of the elements already included in the subsets represented by the node. The inequality below a leaf indicates the reason for its termination.

The state-space tree can be constructed as a binary tree like that in Figure 3 for the instance $A = \{3, 5, 6, 7\}$ and $d = 15$. The root of the tree represents the starting point, with no decisions about the given elements made as yet. Its left and right children represent, respectively, inclusion and exclusion of $a1$ in a set being sought. Similarly, going to the left from a node of the first level corresponds to inclusion of $a2$ while going to the right corresponds to its exclusion, and so on. Thus, a path from the root to a node on the ith level of the tree indicates which of the first I numbers have been included in the subsets represented by that node. We record the value of s, the sum of these numbers, in the node. If s is equal to d, we have a solution to the problem. We can either report this result and stop or, if all the solutions need to be found, continue by backtracking to the node's parent. If s is not equal to d, we can terminate the node as non-promising if either of the following two inequalities holds:

$$s + ai+1 > d \text{ (the sum's is too large),}$$

$$s + = +1 < (h \quad)$$

General Remarks

From a more general perspective, most backtracking algorithms fit the following description. An output of a backtracking algorithm can be thought of as an n-tuple $(x1, x2, \ldots, xn)$ where each coordinate xi is an element of some finite linearly ordered set Si. For example, for the n-queens problem, each Si is the set of integers (column numbers) 1 through n. The tuple may need to satisfy some additional constraints (e.g., the nonattacking requirements in the n-queens problem). Depending on the problem, all solution tuples can be of the same length (the n-queens and the Hamiltonian circuit.

problem) and of different lengths (the subset-sum problem). A backtracking algorithm generates, explicitly or implicitly, a state-space tree; its nodes represent partially constructed tuples with the first i coordinates defined by the earlier actions of the algorithm. If such a tuple $(x1, x2, \ldots, xi)$ is not a solution, the algorithm finds the next element in $Si+1$ that is consistent with the values of $(x1, x2, \ldots, xi)$ and the problem's constraints, and adds it to the tuple as its $(i + 1)$st coordinate. If such an element does not exist, the algorithm backtracks to consider the next value of xi, and so on.

To start a backtracking algorithm, the following pseudocode can be called for$i = 0$; $X[1..0]$ represents the empty tuple.

ALGORITHM *Backtrack(X[1..i])*

//Gives a template of a generic backtracking algorithm

//Input: $X[1..i]$ specifies first i promising components of a solution

//Output: All the tuples representing the problem's solutions

if $X[1..i]$ is a solution **write** $X[1..i]$

else

for each element $x \in Si+1$ consistent with $X[1..i]$ and the constraints **do**

$X[i + 1] \leftarrow x$

Backtrack(X[1..i + 1])

Graph coloring

let G be a graph and m be the a given positive integer. the problem is to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color yet only m colors are used. this is termed as the m-colorability decision problem. Note that if d is the degree o the given graph, then it can be colored with d+ 1 color. The m-color ability optimization problem asks for the smallest integer m for which the graph G can be colored. This integer is referred to as the chromatic number of the graph.

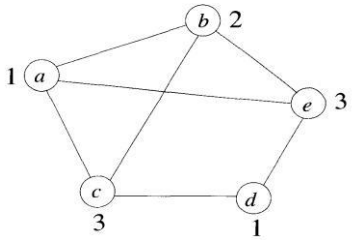

Figure 4 Example graph with its coloring.

The graph in Figure 4 can be colored with three colors 1,2 and 3. The color of each node is indicated next to it. it can also be seen that three colors are needed to color this graph and hence this graph's chromatic number is 3.

Algorithm mcoloring(k)

```
// this algorithm was formed using the recursive backtracking schema. The graph is
//represented by its Boolean adjacency matrix G[1:n,1:n]. All assignments of 1,2,3,,...m
//to the vertices of the graph such that adjacent vertices are assigned distinct integers
//are printed. k is the index of the next vertex to color.

{

repeat

        {

        //generate all legal assignments for x[j].

        nextvalue(k);

        if(x(k)=0) then return;

        if(k=n) then

                write(x[1:n]);

        else mcoloring(k+1);

        } until (false);

}
```

Algorithm for finding m-colorings of the graph

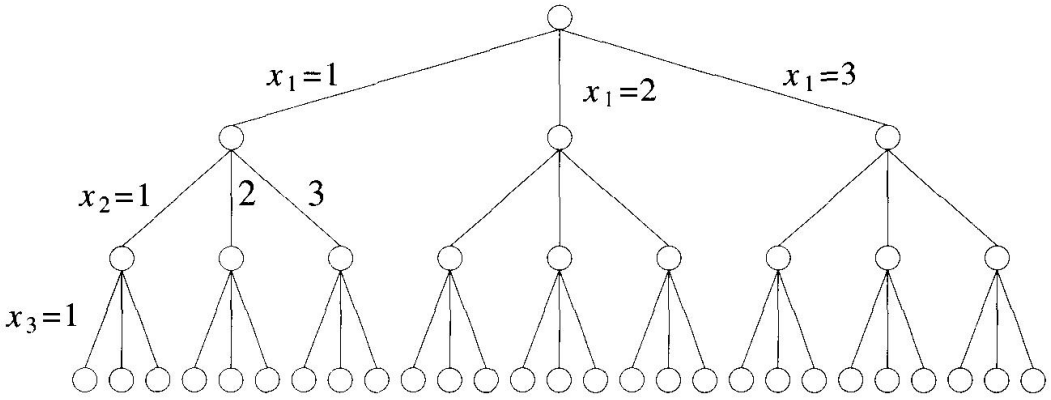

Figure 5.5 State space tree for mcoloring when n=3 and m=3

The underlying state space tree used is a tree of degree m and height n+1. Each node at level I has m children corresponding to the m possible assignments to xi, 1<i<n. nodes at the level n+1 are leaf nodes. Figure 5.5 shows the state space tree when n=3 and m=3.

Hamiltonian cycles

Let G=(V,E) be a connected graph with n vertices. A Hamiltonian cycle is a roundtrip path along n edges of G that visits every vertex once and returns to its starting position. In other words is Hamiltonian cycle begins at some vertex v1∈G and the vertices of G are visited in order $v_1,v_2,v_3,\ldots v_{n+1}$, then the edges$(v_i,v_{i+1})$ are in R, 1<=i<=n, and v_i are distinct except for v_1 and v_{n+1}, which are equal.

The graph G1 of Figure.6 contains the Hamiltonian cycle 1,2,8,7,6,5,4,3,1. The graph G2 of Figure6 contains no Hamiltonian cycle.

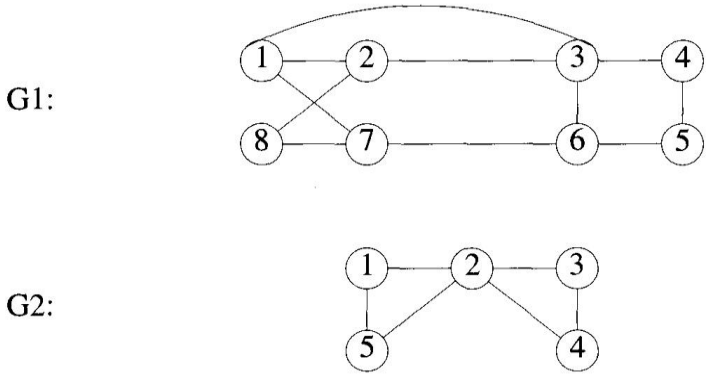

Figure 6: Two graphs, one containing a Hamiltonian cycle.

Backtracking algorithm is used to find all the possible Hamiltonian cycles. The backtracking solution vector(x1,x2….xn) is defines so that xi represents the i^{th} visited vertex of the proposed cycle. Next step is to how to compute the set of possible vertices xk if $x_1,x_2,\ldots x_{k-1}$ have already been chosen. If k=1, then x_1 can be any of the n vertices. To avoid printing the same cycle n times, we require that x_1=1. If 1<k<n, then x_k can be any vertex v that is distinct from $x_1,x_2,\ldots x_{k-1}$ and v is connected by an edge to xk-1. The vertex xn can only be the one remaining vertex and it must be connected to both xn-1 and x1. We begin by presenting function. **nextvalue(k)**, which determines a possible next vertex for the proposed cycle.

Using next value we can particularize the recursive backtracking schema **Hamiltonian(k)** to find all Hamiltonian cycles.

Algorithm nextvalue(k)

//x[1:k-1] is a path of k-1 distinct vertices. If x[k]=0, then no vertex has as yet been //assigned to x[k]. after execution, x[k] is assigned to the next highest numbered vertex /which does not already appear in x[1:k-1] an is connected by an edge to x[k-1]. //Otherwise x[k]=0. If k=n, then in addition x[k] is connected to x[1].

```
{
Repeat
{
        x[k]:=(x[k]+1)mod (n+1);
        if(x[k]=0) then return;
        if(G[x[k-1],x[k])≠0) then
        {
                for j:=1 to k-1 do
                if(x[j]=x[k]) then break;
                if(j=k) then
                if((k<n) or((k=n) and G(x[n],x[1]]≠0))
                then return;
        }
}until (false);
}
```

Algorithm Hamiltonian(k)

//This algorithm uses the recursive formulation of backtracking to find all the //Hamiltonian cycles of a graph.the graph is stored as an adjacency matrix g[1:n-1]. All //cycles begin eith the node 1

```
{ Repeat
{//generate values for x[k]
nextvalue(k);
```

```
if(x[k]=0) then return;

if(k=n) then write(x[1:n]);

else Hamiltonian(k+1);
}until(false);

}
```

Branch and Bound:

the central idea of backtracking, discussed in the previous section, is to cut off a branch of the problem's state-space tree as soon as we can deduce that it cannot lead to a solution. This idea can be strengthened further if we deal with an optimization problem. An optimization problem seeks to minimize or maximize some objective function (a tour length, the value of items selected, the cost of an assignment, and the like), usually subject to some constraints. Note that in the standard terminology of optimization problems, a ***feasible solution*** is a point in the problem's search space that satisfies all the problem's constraints (e.g., a Hamiltonian circuit in the traveling salesman problem or a subset of items whose total weight does not exceed the knapsack's capacity in the knapsack problem), whereas an ***optimal solution*** is a feasible solution with the best value of the objective function (e.g., the shortest Hamiltonian circuit or the most valuable subset of items that fit the knapsack).

Compared to backtracking, branch-and-bound requires two additional items:

- A way to provide, for every node of a state-space tree, a bound on the best value of the objective function on any solution that can be obtained by adding further components to the partially constructed solution represented by the node
- The value of the best solution seen so far

If this information is available, we can compare a node's bound value with the value of the best solution seen so far. If the bound value is not better than the value of the best solution seen so far i.e., not smaller for a minimization problem and not larger for a maximization problem the node is non promising and can be terminated (some people say the branch is "pruned"). Indeed, no solution obtained from it can yield a better solution than the one already available. This is the principal idea of the branch-and-bound technique.

In general, we terminate a search path at the current node in a state-space tree of a branch-and-bound algorithm for any one of the following three reasons:

- The value of the node's bound is not better than the value of the best solution seen so far.

- The node represents no feasible solutions because the constraints of the problem are already violated.
- The subset of feasible solutions represented by the node consists of a single point(and hence no further choices can be made)—in this case, we compare the value of the objective function for this feasible solution with that of the best solution seen so far and update the latter with the former if the new solution is better

Assignment Problem

Let us illustrate the branch-and-bound approach by applying it to the problem of assigning n people to n jobs so that the total cost of the assignment is as small as possible. Recall that an instance of the assignment problem is specified by an $n \times n$ cost matrix C so that we can state the problem as follows: select one element in each row of the matrix so that no two selected elements are in the same column and their sum is the smallest possible.

$$C = \begin{array}{c} \begin{matrix} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \end{matrix} \\ \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \end{array} \begin{matrix} \\ \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{matrix}$$

How can we find a lower bound on the cost of an optimal selection without actually solving the problem? We can do this by several methods. For example, it is clear that the cost of any solution, including an optimal one, cannot be smaller than the sum of the smallest elements in each of the matrix's rows. For the instance here, this sum is 2 + 3+ 1+ 4 = 10. It is important to stress that this is not the cost of any legitimate selection (3 and 1 came from the same column of the matrix); it is just a lower bound on the cost of any legitimate selection. We can and will apply the same thinking to partially constructed solutions. For example, for any legitimate selection that selects 9 from the first row, the lower bound will be 9 + 3 + 1+ 4 = 17.

One more comment is in order before we embark on constructing the problem's state-space tree. It deals with the order in which the tree nodes will be generated. Rather than generating a single child of the last promising node as we did in backtracking, we will generate all the children of the most promising node among non terminated leaves in the current tree. (Non terminated, i.e., still promising, leaves are also called ***live***.) How can we tell which of the nodes is most promising? We can do this by comparing the lower bounds of the live nodes. It is sensible to consider a node with the best bound as most promising, although this does not, of course, preclude the possibility that an optimal solution will ultimately belong to a different branch of the state-space tree. This variation of the strategy is called the ***best-first branch-and-bound***.

So, returning to the instance of the assignment problem given earlier, we start with the root that corresponds to no elements selected from the cost matrix. As we already discussed, the lower-bound value for the root, denoted *lb*, is 10. The nodes on the first level

of the tree correspond to selections of an element in the first row of the matrix, i.e., a job for person *a* (Figure 7).

So we have four live leaves—nodes 1 through 4—that may contain an optimal solution. The most promising of them is node 2 because it has the smallest lower bound value. Following our best-first search strategy, we branch out from that node first by considering the three different ways of selecting an element from the second row and not in the second column—the three different jobs that can be assigned to person *b* (Figure 8).

Of the six live leaves—nodes 1, 3, 4, 5, 6, and 7—that may contain an optimal solution, we again choose the one with the smallest lower bound, node 5. First, we consider selecting the third column's element from *c*'s row (i.e., assigning person *c* to job 3); this leaves us with no choice but to select the element from the fourth column of *d*'s row (assigning person *d* to job 4). This yields leaf 8 (Figure 9), which corresponds to the feasible solution {*a*→2, *b*→1, *c*→3, *d* →4} with the total cost of 13. Its sibling, node 9, corresponds to the feasible solution {*a*→2, *b*→1, *c*→4, *d* →3} with the total cost of 25. Since its cost is larger than the cost of the solution represented by leaf 8, node 9 is simply terminated. (Of course, if its cost were smaller than 13, we would have to replace the information about the best solution seen so far with the data provided by this node.)

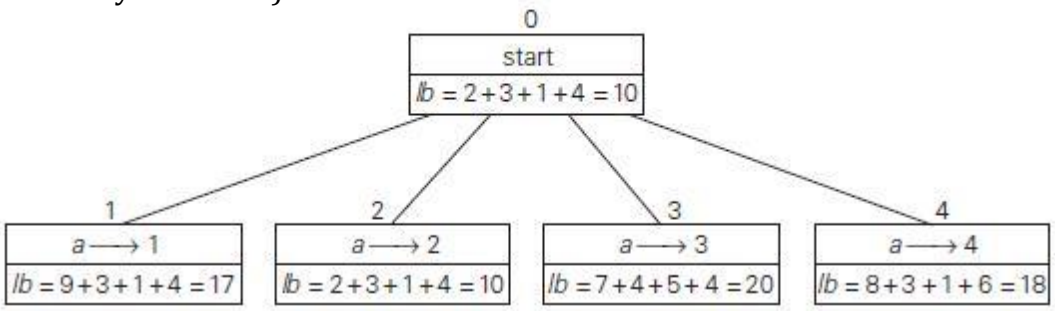

FIGURE 7 Levels 0 and 1 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm. The number above a node shows the order in which the node was generated.

A node's fields indicate the job number assigned to person *a* and the lower bound value, *lb,* for this node.

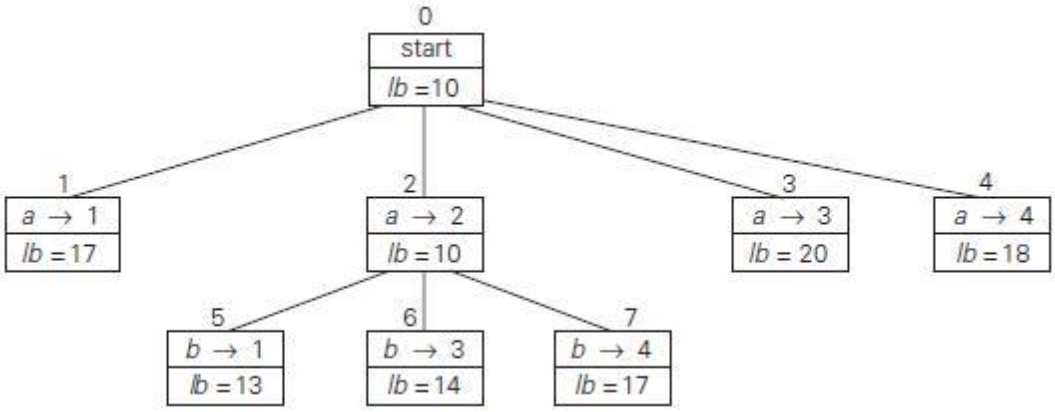

FIGURE 8 Levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm

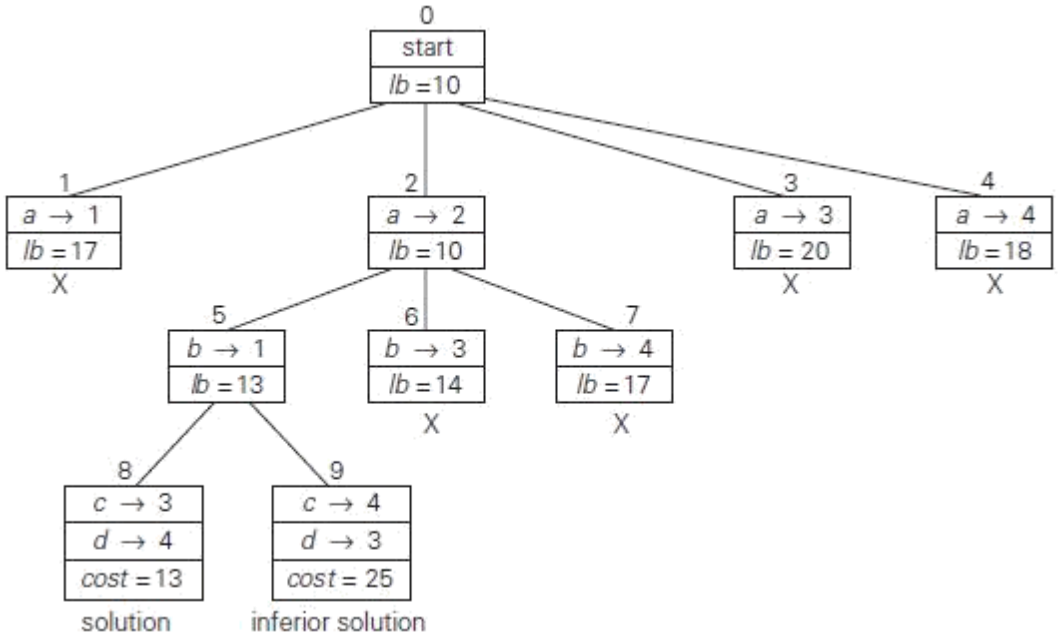

FIGURE 9 Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm.

Now, as we inspect each of the live leaves of the last state-space tree—nodes 1, 3, 4, 6, and 7 in Figure 9—we discover that their lower-bound values are not smaller than 13, the value of the best selection seen so far (leaf 8). Hence, we terminate all of them and recognize the solution represented by leaf 8 as the optimal solution to the problem.

Traveling Salesman Problem

The branch-and-bound technique to instances of the traveling salesman problem if we come up with a reasonable lower bound on tour lengths. One very simple lower bound can be obtained by finding the smallest element in the intercity distance matrix *D* and multiplying it by the number of cities *n.* But there is a less obvious and more informative lower bound for instances with symmetric matrix *D*, which does not require a lot of work to compute. It is not difficult to show that we can compute a lower bound on the length *l* of any tour as follows. For each city *i*, 1≤ *i* ≤ *n*, find the sum *si* of the distances from city *i* to the two nearest cities; compute the sums of these *n* numbers, divide the result by 2, and, if all the distances are integers,
round up the result to the nearest integer:

$$lb = s/2. \qquad (1)$$

For example, for the instance in Figure 10a, formula (1) yields *lb* = _[*(*1+ 3*)* + *(*3 + 6*)* + *(*1+ 2*)*

+ *(3 + 4) + (2 + 3)]/2_ = 14.*Moreover, for any subset of tours that must include particular edges of a given graph, we can modify lower bound (1) accordingly. For example, for all the Hamiltonian circuits of the graph in Figure 5.10 a that must include edge *(a, d)*,we get the following lower bound by summing up the lengths of the two

Shortest edges incident with each of the vertices, with the required inclusion of edges *(a, d)* and *(d, a)*:[*(*1+ 5*) + (*3 + 6*) + (*1+ 2*) + (*3 + 5*) + (*2 + 3*)]/2 = 16.*

We now apply the branch-and-bound algorithm, with the bounding function given by formula (1), to find the shortest Hamiltonian circuit for the graph in

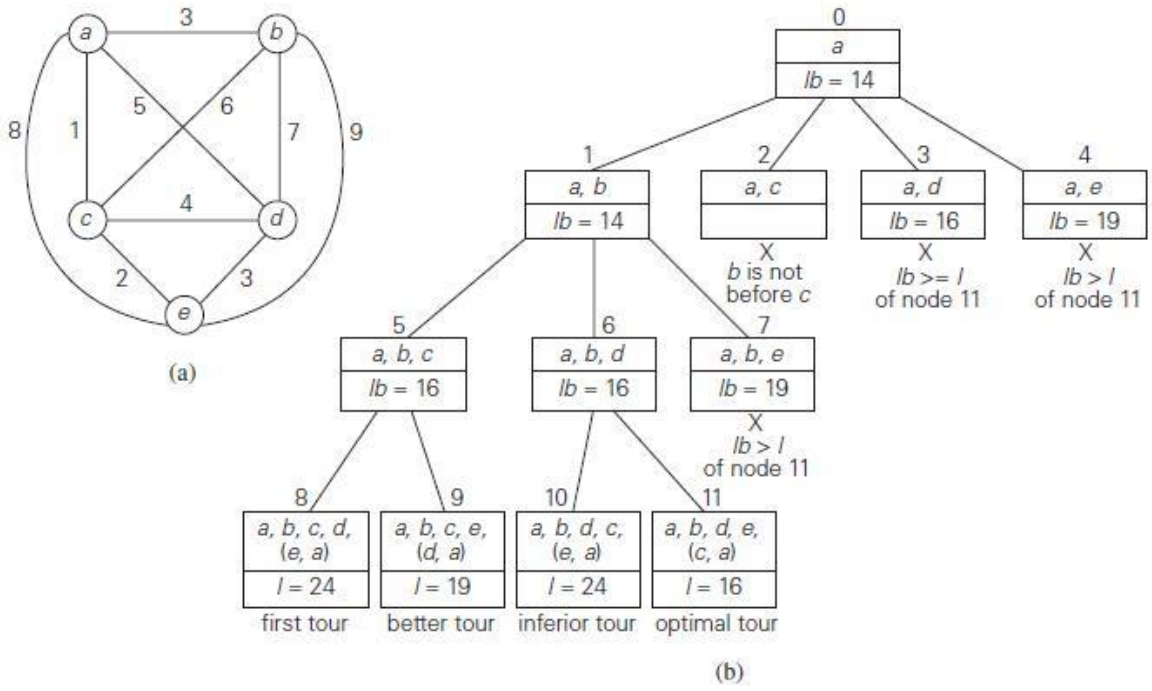

FIGURE 10 (a)Weighted graph. (b) State-space tree of the branch-and-bound algorithm.

To find a shortest Hamiltonian circuit in this graph. The list of vertices in a node specifies a beginning part of the Hamiltonian circuits represented by the node. Figure 10a. To reduce the amount of potential work, we take advantage of two observations made in First, without loss of generality, we can consider only tours that start at *a*. Second, because our graph is undirected, we can generate only tours in which *b* is visited before *c.* In addition, after visiting *n* − 1= 4 cities, a tour has no choice but to visit the remaining unvisited city and return to the starting one. The state-space tree tracing the algorithm's application is given in Figure 10b.

0/1 Knapsack problem

The branch-and-bound technique to solving the knapsack problem. given *n* items of known weights *wi* and values *vi* , *i* = 1, 2, . . . , *n,* and a knapsack of capacity *W*, find the most

valuable subset of the items that fit in the knapsack. It is convenient to order the items of a given instance in descending order by their value-to-weight ratios. Then the first item gives the best payoff per weight unit and the last one gives the worst payoff per weight unit, with ties resolved arbitrarily:

$v1/w1 \geq v2/w2 \geq \ldots \geq vn/wn.$

It is natural to structure the state-space tree for this problem as a binary tree constructed as follows (see Figure 11 for an example). Each node on the ith level of this tree, $0 \leq i \leq n$, represents all the subsets of n items that include a particular selection made from the first i ordered items. This particular selection is uniquely determined by the path from the root to the node: a branch going to the left indicates the inclusion of the next item, and a branch going to the right indicates its exclusion. We record the total weight w and the total value v of this selection in the node, along with some upper bound ub on the value of any subset that can be obtained by adding zero or more items to this selection.

A simple way to compute the upper bound ub is to add to v, the total value of the items already selected, the product of the remaining capacity of the knapsack $W - w$ and the best per unit payoff among the remaining items, which is $vi+1/wi+1$: $ub = v + (W - w)(vi+1/wi+1)$. **(2)**

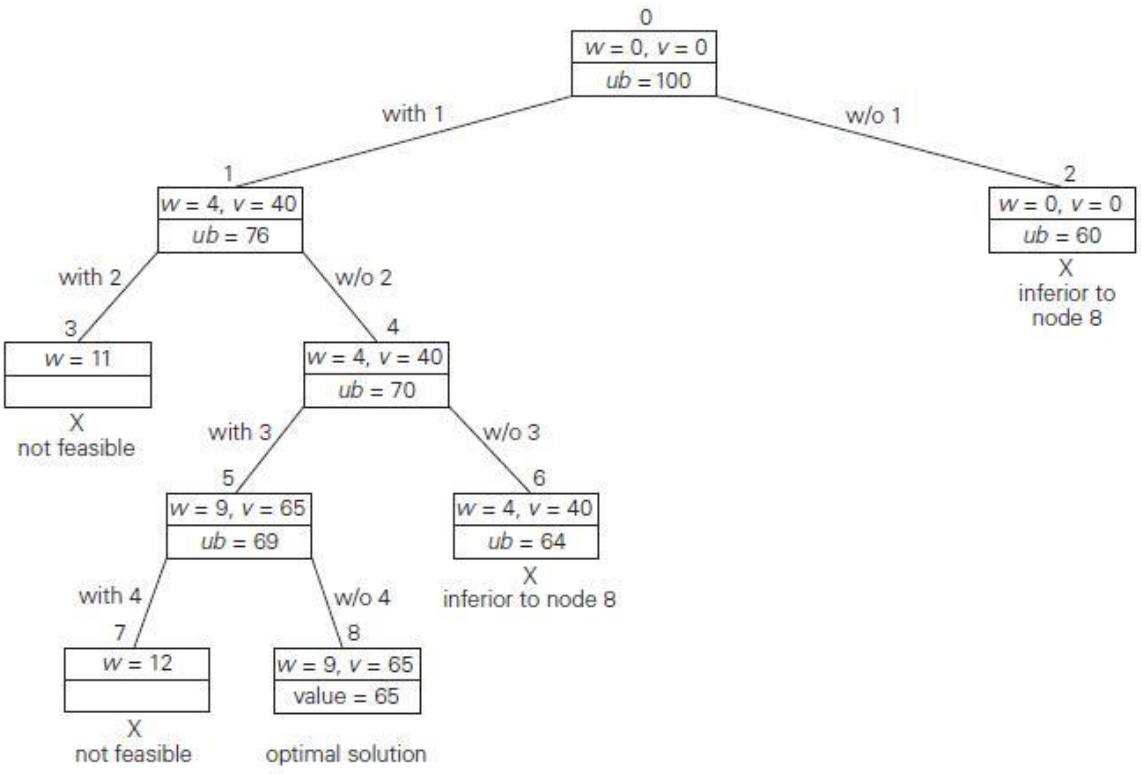

FIGURE 11 State-space tree of the best-first branch-and-bound algorithm for the instance of the knapsack problem.

At the root of the state-space tree (see Figure 5.11), no items have been selected as yet. Hence, both the total weight of the items already selected w and their total value v are equal

to 0. The value of the upper bound computed by formula (1) is \$100. Node 1, the left child of the root, represents the subsets that include item 1. The total weight and value of the items already included are 4 and \$40, respectively; the value of the upper bound is $40 + (10 - 4) * 6 = \$76$. Node 2 represents the subsets that do not include item 1. Accordingly, $w = 0$, $v = \$0$, and $ub = 0 + (10 - 0) * 6 = \$60$. Since node 1 has a larger upper bound than the upper bound of node 2, it is more promising for this maximization problem, and we branch from node 1 first. Its children—nodes 3 and 4—represent subsets with item 1 and with and without item 2, respectively. Since the total weight w of every subset represented by node 3 exceeds the knapsack's capacity, node 3 can be terminated immediately. Node 4 has the same values of w and v as its parent; the upper bound ub is equal to $40 + (10 - 4) * 5 = \$70$. Selecting node 4 over node 2 for the next branching (why?), we get nodes 5 and 6 by respectively including and excluding item 3. The total weights and values as well as the upper bounds for these nodes are computed in the same way as for the preceding nodes. Branching from node 5 yields node 7, which represents no feasible solutions, and node 8, which represents just a single subset {1, 3} of value \$65. The remaining live nodes 2 and 6 have smaller upper-bound values than the value of the solution represented by node 8. Hence, both can be terminated making the subset {1, 3} of node 8 the optimal solution to the problem.

NP-Complete and NP-Hard problems

Basic concepts

Concerned with the distinction between problems that can be solved by the polynomial time algorithm and problems for which no polynomial time algorithm is known. For many problems that we studies, the best algorithm for their solutions have computing times that cluster into two groups.

The first group consists of problems whose times are bounded by polynomial of small degree. Examples like searching which is O(logn), polynomial evaluation which is O(n), sorting which is O(nlogn), and string editing which is O(mn).

The second group is made up of problems whose best-known algorithms are non polynomial. Examples like travelling sales person problem which is with the complexity $O(n^2 2^n)$ and knapsack problem with $O(2^{n/2})$.

Nondeterministic algorithms

Notion of the algorithm that we have been using has the property that the result of every operation is uniquely defines. Algorithms with this property are termed deterministic algorithms. Such algorithms agree with the way programs are executed on a computer. in a theoretical framework we can remove this restriction on the outcome of every operation . We can allow the algorithms to contain operations whose outcomes are not uniquely defines but are limited to specified sets of possibilities. The machine executing such operations is allowed to choose any one of these outcomes subject to a termination condition to be defines later. This leads to the concept of a nondeterministic algorithm. To specify such algorithms, we introduce

three new functions:

1. Choice(S) arbitrarily chooses one of the elements of set S.
2. Failure() signals an unsuccessful completion
3. Success () signals a successful completion.

The assignment statement x:=choice(1,n) could result in x being assigned any one of the integers in the range[1,n]. There is no rule specifying how this choice is to be made. The failure () and success () signals are used to define a computation of the algorithm. These statements cannot be used to effect a return. Whenever there is a set of choices that leads to a successful completion, then one such set of choices is always made and the algorithm terminates successfully. A nondeterministic algorithm terminates unsuccessfully if and only of there exists no set of choices leading to a success signal. The computing times of success, choice, failure are taken to be O(1).

The classes P, NP-HARD and NP-complete

In measuring the complexity of an algorithm, we use the input length as the parameter. An algorithm A is of polynomial complexity if there exists a polynomial p() such that this computing time of A is O(p(n)) for every input of size n.

Definition

P is the set of all decision problems solvable by deterministic algorithms in polynomial time. **NP** is the set of all decision problems solvable by nondeterministic algorithms in polynomial time.

Since deterministic algorithms are just a special case of nondeterministic ones, we conclude that P⊆NP.

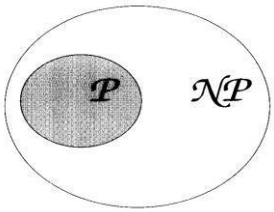

Figure 14 commonly believed relationship between P and NP

Definition

Let L1 and L2 be problems. Problem L1 reduces to L2 if and only if there is a way to solve L1 by a deterministic polynomial time algorithm using a deterministic algorithm that solves L2 in polynomial time.

Definition

A problem L is **NP –hard** if and only if satisfiability reduces to L. A problem L is **NP-complete** if and only if L is NP-hard and L∈NP

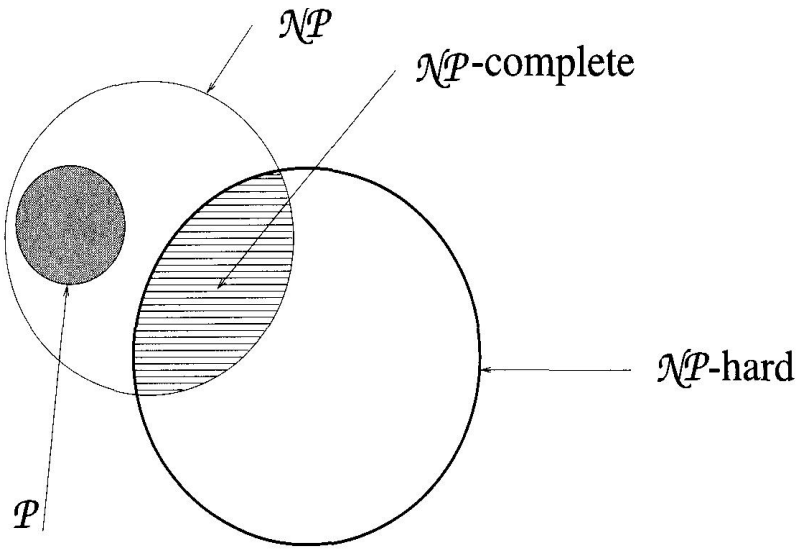

Figure 15 commonly believed relationship among P, NP, NP-complete and NP-hard problems

Definition

Two problems L1 and L2 are said to be polynomially equivalent if and only If L1 reduces to L2 and L2 reduces to L1